Corrigé DS4

Ce corrigé concerne l'union des deux sujets. Les questions peuvent porter deux numéros avec les conventions suivantes : si le numéro de la question est 2) - 3*) cela signifie que cette question avait le numéro 2 dans le DS4 et le numéro 3 dans le DS4*.

1) a) On a $f(1,2) = 3$ par lecture de la figure 2, $f(3,4) = -f(4,3) = -1$ par antisymétrie et $f(1,4) = 0$. Cette dernière égalité se déduit de :

   - Le respect de la capacité impose $f(1,4) \leq c(1,4) = 0$ car l'arête $(1,4)$ n'existe pas.
   - De même, le respect de la capacité impose que $f(4,1) \leq 0$. Or, l'antisymétrie nous dit que $f(4,1) = -f(1,4)$ et on en déduit que $f(1,4) \geq 0$.

   b) Le débit du flot vaut par définition $f(s,1) + f(s,2) = 5$, les autres valeurs de $f(s,u)$ étant nulles d'après le même raisonnement que celui tenu pour déterminer $f(1,4)$ à la question précédente.

   c) La lecture de la figure 2 assure que $\varphi_-(t) = 5$, $\varphi_+(2) = 5$ et $\varphi(3) = 0 + 1 - 1 = 0$.

2) - 1*) Si $f(u,v) > 0$, supposons par l'absurde que $(u,v) \notin A$. Alors, $c(u,v) = 0$ et le respect de la capacité impose que $0 < f(u,v) \leq c(u,v) = 0$ ce qui n'est pas. Donc $(u,v) \in A$.

   De plus, $f(u,v) < 0$ revient à $f(v,u) > 0$ par antisymétrie donc le raisonnement précédent montre que $(v,u) \in A$ dans ce cas.

3) - 2*) On a pour tout $u \in S \setminus \{s,t\}$:

$$\begin{aligned}
\varphi(u) &= \varphi_+(u) - \varphi_-(u) \\
&= \sum_{\substack{v \in S \\ f(u,v)>0}} f(u,v) - \sum_{\substack{v \in S \\ f(u,v)<0}} f(v,u) \quad \text{par définition} \\
&= \sum_{\substack{v \in S \\ f(u,v)>0}} f(u,v) + \sum_{\substack{v \in S \\ f(u,v)<0}} f(u,v) \quad \text{par antisymétrie} \\
&= \sum_{\substack{v \in S \\ f(u,v)>0}} f(u,v) + \sum_{\substack{v \in S \\ f(u,v)<0}} f(u,v) + \sum_{\substack{v \in S \\ f(u,v)=0}} f(u,v) \quad \text{car on rajoute des termes nuls} \\
&= \sum_{v \in S} f(u,v) = 0 \quad \text{par conservation}
\end{aligned}$$

   Cela montre que la conservation du flot est équivalente au fait que le flux de tout sommet différent de s et t est nul. Cette dernière propriété s'exprime en français de la façon suivante : rien ne se créé, rien ne se perd dans aucun sommet qui n'est ni la source ni le puits ; tout ce qui entre dans un tel sommet doit en sortir.

4) - 3*) On a d'une part :

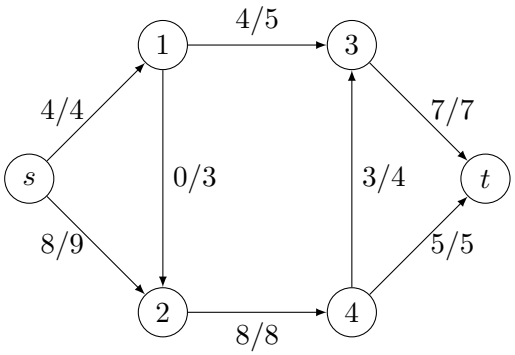
$$|f| = \sum_{v \in S} f(s,v) = \varphi_+(s) - \varphi_-(s) = \varphi(s)$$

   en décomposant la somme de la même manière qu'en question 3. D'autre part, la question précédente assure que $\sum_{u \in S} \varphi(u) = \varphi(s) + \varphi(t)$ puisque tous les autres termes de cette somme sont nuls. Donc :

$$\begin{aligned}
\varphi(s) + \varphi(t) &= \sum_{u \in S} \varphi(u) = \sum_{u \in S} \varphi_+(u) - \sum_{u \in S} \varphi_-(u) \\
&= \sum_{\substack{(u,v) \in S^2 \\ f(u,v) > 0}} f(u,v) - \sum_{\substack{(u,v) \in S^2 \\ f(u,v) < 0}} f(v,u) \quad \text{par définition des flux} \\
&= \sum_{\substack{(u,v) \in S^2 \\ f(u,v) > 0}} f(u,v) - \sum_{\substack{(u,v) \in S^2 \\ f(v,u) > 0}} f(v,u) \quad \text{par antisymétrie} \\
&= 0 \quad \text{par réindiçage}
\end{aligned}$$

On en déduit que $\varphi(t) = -\varphi(s) = -|f|$. Ceci est intuitif : si on créé X entités en s et qu'aucun sommet autre que s et t ne créé ni ne prélève d'entité, alors X entités arrivent en t.

5) - 4*) On propose le flot suivant :



Il respecte bien les capacités et le fait que le flux net des sommets internes est nul : c'est donc bien un flot. Par ailleurs, $|f| = -\varphi(t) = 12$. Un flot de débit strictement supérieur entraînerait un non respect de la capacité d'une des arêtes entrantes de t : le flot est donc maximal.

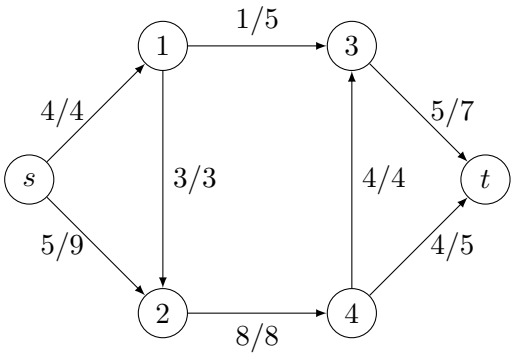
6) - 5*) On peut effectuer un parcours de graphe depuis s en ne s'autorisant à emprunter que les arêtes non saturées et en stockant l'arbre de parcours. Si le sommet t est atteint lors de ce parcours, alors c'est via un chemin non saturé qu'on peut reconstruire via l'arbre de parcours. Sinon, cela signifie que tous les chemins de s à t empruntent au moins une arête saturée (donc de capacité disponible nulle) et comme la capacité disponible d'un chemin est le minimum des capacités disponibles de ses arêtes, cela signifie que tous les chemins de s à t sont saturés.

Le coût du parcours est de l'ordre de $O(|S| + |A|)$ et le coût de la reconstruction d'un éventuel chemin de s à t en $O(|S|)$ si on stocke le père de chaque sommet dans l'arbre de parcours dans un tableau de taille $|S|$. Le coût total pour déterminer si un chemin non saturé existe est donc en $O(|S| + |A|)$.

7) - 6*) L'opération de saturation préserve l'antisymétrie, le respect de la capacité et la conservation donc le résultat d'une telle opération reste un flot. De plus, si (u, v) est l'arête de capacité disponible minimale sur le chemin C, après saturation sa capacité disponible vaut 0 par construction donc C est saturé.

Le nombre de chemins non saturés entre s et t est fini et décroît strictement à chaque itération de la boucle tant que d'après ce qui précède d'où la terminaison de flot_glouton. Comme la fonction identiquement nulle est un flot, "f est un flot sur G" est un invariant pour la boucle tant que de flot_glouton d'après ce qui précède donc cet algorithme renvoie bien un flot.

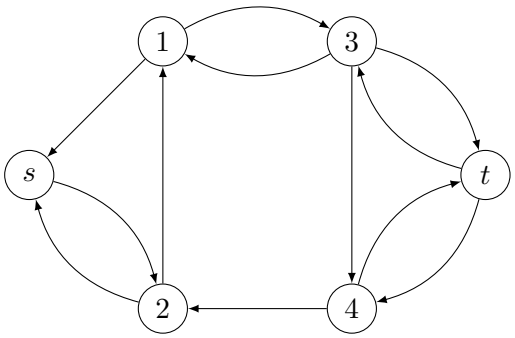
8) - 7*) Le seul chemin non saturé restant est $(s, 1, 3, t)$, de capacité disponible 1. Une fois saturé, on obtient le flot suivant :



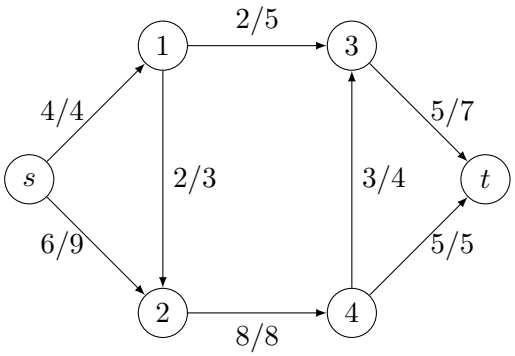
A présent, tous les chemins sont saturés, et on obtient donc un flot dont le débit est $9 < 12$. D'après la question 5, ce flot n'est pas maximal.

9) - 8*) Pour que (u, v) appartienne à A_f, il est nécessaire (mais non suffisant) que $c(u, v) > 0$ ou $f(u, v) < 0$. La première condition implique que $(u, v) \in A$ et la deuxième que $(v, u) \in A$ d'après la question 2.

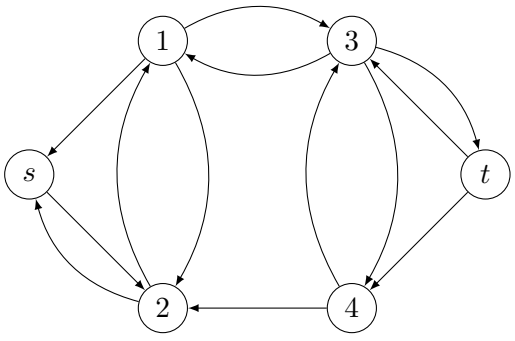
10) - 9*) On obtient :



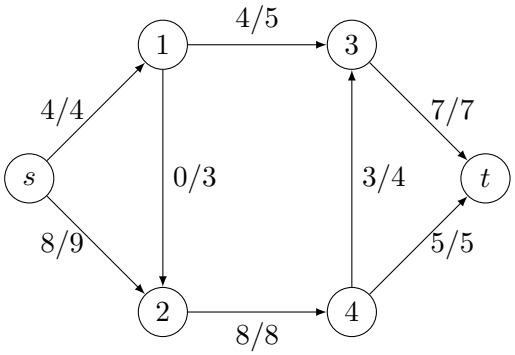
11) - 10*) On constate que dans le graphe précédent $(s, 2, 1, 3, 4, t)$ améliorant. Après l'avoir saturé, on obtient le flot suivant :



Le graphe résiduel associé à ce flot est :



Ainsi, le chemin $(s, 2, 1, 3, t)$ est améliorant et en le saturant, on obtient un flot dont le graphe résiduel ne contient aucun chemin. L'algorithme de Ford-Fulkerson renvoie donc le flot suivant, qui cette fois est maximal d'après la question 5 :



12) Immédiat.

```
maillon* creer_maillon(int v)
{
    maillon* m = (maillon*) malloc(sizeof(maillon));
    m->donnee = v;
    m->suivant = NULL;
    return m;
}
```

13) Immédiat.

```
file* creer_file(void)
{
    file* f = (file*)malloc(sizeof(file));
    f->debut = NULL;
    f->fin = NULL;
    f->longueur = 0;
    return f;
}
```

14) Encore plus immédiat.

```
int longueur_file(file* f)
{ return f->longueur; }
```

15) On traite le cas d'une file vide à part comme suggéré par l'énoncé.

```
void ajouter_file(file* f, int v)
{
    maillon* m = creer_maillon(v);
    if (longueur_file(f) == 0)
    {
        f->debut = m;
        f->fin = m;
    }
    else
```

```
    {
        f->fin->suivant = m;
        f->fin = m;
    }
    f->longueur++;
}
```

16) Si la longueur de la file vaut un, il faut correctement modifier les champs debut et fin pour maintenir l'invariant imposé par l'énoncé. Ne pas oublier de libérer le maillon supprimé... au bon moment.

```
int extraire_file(file* f)
{
    assert(longueur_file(f) > 0);
    int valeur_tete = f->debut->donnee;
    if (longueur_file(f) == 1)
    {
        free(f->debut);
        f->debut = NULL;
        f->fin = NULL;
    }
    else
    {
        maillon* nouvelle_tete = f->debut->suivant;
        free(f->debut);
        f->debut = nouvelle_tete;
    }
    f->longueur--;
    return valeur_tete;
}
```

17) Immédiat. Attention à l'ordre des libérations, comme toujours.

```
void liberer_maillon(maillon* m)
{
    if (m != NULL)
    {
        liberer_maillon(m->suivant);
        free(m);
    }
}
```

```
void liberer_file(file* f)
{
    liberer_maillon(f->debut);
    free(f);
}
```

18) Insérer en tête de file ne pose pas de problème avec cette structure, ce pourrait être fait en $O(1)$. L'extraction en queue de file est en revanche problématique car il faudrait modifier fin de sorte qu'il pointe vers l'avant-dernier maillon. Pour récupérer un pointeur vers ledit maillon, il n'y a pas d'autre possibilité que de parcourir toute la file depuis le début ce qui coûte un $O(n)$ où n est la longueur de la file.

19) - 11*) Immédiat.

```
int** zeros(int n)
{
    int** matrice = (int**)malloc(sizeof(int*)*n);
    for (int i = 0; i < n; i++)
    {
        matrice[i] = (int*)malloc(sizeof(int)*n);
        for (int j = 0; j < n; j++)
        {
            matrice[i][j] = 0;
        }
    }
    return matrice;
}
```

20) - 12*) Immédiat.

```
void liberer_matrice(int** m, int n)
{
    for (int i = 0; i < n; i++) { free(m[i]); }
    free(m);
}
```

21) - 13*) Il est inutile de construire explicitement le graphe résiduel : un arc de ce graphe est soit un arc (u, v) du graphe initial, soit un arc "renversé" mais la condition en gras dans l'énoncé nous assure que tous les arcs renversés sont présent dans le graphe initial, avec une capacité nulle. Les arcs du graphe résiduel sont donc ceux du graphe en entrée pour lesquels la différence entre la capacité et le flot est strictement positive : il suffit de filtrer les arcs du graphe initial selon ce critère pour obtenir les arcs du graphe résiduel.

On utilise le tableau parents pour deux choses : stocker l'arbre de parcours et détecter si un sommet à déjà été visité. En effet, parents[i] vaut -1 si et seulement si i n'a pas encore été visité d'après l'initialisation de parents.

Autrement, cette fonction est un parcours en largeur classique (et on comprend mieux à présent pourquoi l'énoncé nous faisait implémenter une file, qu'on n'oublie pas de libérer avant la fin de la fonction).

```c
int* BFS_residuel(graphe_flot* g, int** f)
{
    int n = g->n;
    int* parents = (int*)malloc(sizeof(int)*n);
    for (int i = 0; i < n; i++) parents[i] = -1;
    file* a_visiter = creer_file();

    ajouter_file(a_visiter,g->s);
    parents[g->s] = g->s;

    while(longueur_file(a_visiter) > 0)
    {
        int v = extraire_file(a_visiter);
        for (int i = 0; i < g->degres[v]; i++)
        {
            int w = g->voisins[v][i];
            if (parents[w] == -1 && g->capacites[v][w] - f[v][w] > 0)
            {
                parents[w] = v;
                ajouter_file(a_visiter,w);
            }
        }
    }
    liberer_file(a_visiter);
    return parents;
}
```

22) - 14*) On commence par vérifier que t a un parent, si non, il n'est pas accessible depuis s. S'il en a un, on remonte le chemin entre s et t depuis t graĉe à l'arbre de parcours et pour chaque arc sur ce chemin, on met à jour la capacité disponible de ce dernier. Cette capacité disponible est initialisée avec le plus grand entier possible INT_MAX, neutre pour min.

```c
int capacite_chemin(graphe_flot* g, int**f, int* parents)
{
    int capacite_dispo = INT_MAX;
    int v = g->t;
    if (parents[v] == -1) { return 0; }
    while (v != g->s)
    {
        int p = parents[v];
        capacite_dispo = min(capacite_dispo, g->capacites[p][v] - f[p][v]);
        v = p;
    }
    return capacite_dispo;
}
```

23) - 15*) Le principe de cette fonction est quasiment identique à celui de la précédente, à l'exception du fait que le traitement de chaque arc sur le chemin exige deux opérations : celles décrites par l'opération de saturation en partie 2.

```
bool saturer_chemin(graphe_flot* g, int** f, int* parents)
{
    int capacite_dispo = capacite_chemin(g,f,parents);
    if (capacite_dispo == 0) { return false; }
    int v = g->t;
    while(v != g->s)
    {
        int p = parents[v];
        f[p][v] = f[p][v] + capacite_dispo;
        f[v][p] = f[v][p] - capacite_dispo;
        v = p;
    }
    return true;
}
```

24) - 16*) On applique successivement BFS_residuel puis saturer_chemin sans oublier de libérer la mémoire allouée sur le tas lors de l'appel à BFS_residuel.

```
bool etape(graphe_flot* g, int** f)
{
    int* parents = BFS_residuel(g,f);
    bool res = saturer_chemin(g,f,parents);
    free(parents);
    return res;
}
```

25) - 17*) Dans BFS_residuel, chaque sommet est inséré dans la file au plus une fois et donc extrait au plus une fois. Le coût de traitement d'un sommet lors de son extraction est proportionnel à son degré sortant dans le graphe résiduel (d'après la complexité des opérations sur les files) ; or ce dernier est au plus le double du degré sortant de ce sommet dans le graphe initial. La boucle while s'exécute donc en temps $O\left(\sum_{v\in S} \deg_+(v)\right) = O(|A|)$. En ajoutant à ce coût la création du tableau parents, on obtient une complexité en $O(|S| + |A|)$.

La fonction saturer_chemin demande de parcourir deux fois (un fois pour déterminer sa capacité disponible et une fois pour le saturer) un chemin de taille au plus $|S|$ donc s'exécute en $O(|S|)$. Ainsi, la complexité de etape est en $O(|S| + |A| + |S|) = O(|S| + |A|)$.

26) - 18*) Tout le travail a déjà été fait par etape :

```
int** ford_fulkerson(graphe_flot* g)
{
    int** f = zeros(g->n);
    while (etape(g,f)) {}
    return f;
}
```

27) - 19*) La saturation d'un chemin améliorant augmente le débit du flot d'une valeur entière strictement positive. Comme ce dernier est borné (par la somme des capacités des arcs sortants de s), l'algorithme termine.

Ceci montre également qu'on appelera la fonction etape au plus $O(M)$ fois donc que le coût de la boucle tant que de l'algorithme précédent est en $O(M(|S| + |A|))$ d'après la question 25. A ceci s'ajoute le coût de la création du flot inital en $O(|S|^2)$ d'où une complexité totale en $O(|S|^2 + M(|S| + |A|))$.

20*) On a $C(X) = c(s,2) + c(1,2) + c(3,t) = 19$.

21*) D'après les questions 2* et 3*, on a $|f| = \varphi(s) = \sum_{u \in X} \varphi(u)$. D'autre part

$$\begin{aligned}
\sum_{u \in X} \varphi(u) &= \sum_{u \in X} (\varphi_+(u) - \varphi_-(u)) \\
&= \sum_{u \in X} \left(\sum_{\substack{v \in X \\ f(u,v)>0}} f(u,v) + \sum_{\substack{v \in \overline{X} \\ f(u,v)>0}} f(u,v) - \sum_{\substack{v \in X \\ f(u,v)<0}} f(v,u) - \sum_{\substack{v \in \overline{X} \\ f(u,v)<0}} f(v,u) \right) \\
&= \sum_{u \in X} \left(\sum_{\substack{v \in \overline{X} \\ f(u,v)>0}} f(u,v) + \sum_{\substack{v \in \overline{X} \\ f(u,v)<0}} f(u,v) \right) + \sum_{\substack{u,v \in X \\ f(u,v)>0}} f(u,v) - \sum_{\substack{u,v \in X \\ f(u,v)<0}} f(v,u) \\
&= \sum_{(u,v) \in X \times \overline{X}} f(u,v) + \underbrace{\sum_{\substack{u,v \in X \\ f(u,v)>0}} f(u,v) - \sum_{\substack{u,v \in X \\ f(v,u)>0}} f(v,u)}_{=0}
\end{aligned}$$

Comme le flot f respecte les capacités on a donc :

$$|f| = \sum_{(u,v) \in X \times \overline{X}} f(u,v) \leq \sum_{(u,v) \in X \times \overline{X}} c(u,v) = C(X)$$

22*) (1) $\Rightarrow$ (2) S'il existait un chemin améliorant, on pourrait augmenter strictement le débit du flot en le saturant, c'est une contradiction avec (1).

(2) $\Rightarrow$ (3) Posons X l'ensemble des sommets accessibles depuis s dans le graphe résiduel G_f. Comme il n'y a pas de chemin améliorant, $t \notin X$ donc X est une coupe. De plus, les arcs (u,v) tels que $u \in X$ et $v \notin X$ ne sont pas dans G_f sinon v serait accessible depuis s donc pour un tel arc on a $c(u,v) = f(u,v)$. D'après la question précédente,

$$|f| = \sum_{(u,v) \in X \times \overline{X}} f(u,v) = \sum_{(u,v) \in X \times \overline{X}} c(u,v) = C(X)$$

et l'existence d'une coupe convenable est donc démontrée.

(3) $\Rightarrow$ (1) Considérons un autre flot f'. La question 21*) montre que $|f'| \leq C(X)$ où X est la coupe fournie par (2). On a donc $|f'| \leq |f|$ ce qui montre que f est maximal.

Remarque : Ceci est le théorème coupe-max / flot-min (max flow / min cut theorem).

23*) L'algorithme termine d'après la question 19*) et renvoie un flot dont le graphe résiduel ne contient pas de chemin améliorant. L'implication (2) $\Rightarrow$ (1) prouvée à la question 22*) permet de conclure que ce flot est maximal.

24*) Soit $G = (X \sqcup Y, A)$ un graphe biparti. On définit le graphe de flot $G' = (S, A', c, s, t)$ par :

- $S = X \sqcup Y \cup \{s\} \cup \{t\}$ où s et t sont deux nouveaux sommets.
- Pour tout $x \in X$ et tout $y \in Y$, $(s,x) \in A'$ et $(y,t) \in A'$.
- Pour tout $(x,y) \in X \times Y$, $(x,y) \in A'$ si et seulement si $\{x,y\} \in A$; autrement dit, on oriente les arêtes de G de X vers Y.
- c est la constante égale à 1 sur A'.

Si f est un flot sur G', on note $C_f = \{\{x,y\} \in A \,|\, f(x,y) = 1\}$. Si $x_0, y_0, ..., x_k, y_k$ est un chemin C_f-augmentant alors pour tout i, $f(x_i, y_i) = 0$ et $f(y_i, x_{i+1}) = -1$. Comme les capacités valent toutes un, on en déduit que $s, x_0, y_0, ...x_k, y_k, t$ est un chemin améliorant pour f. Réciproquement, en supprimant s et t d'un chemin améliorant pour f, on obtient un chemin C_f-augmentant.

D'après le lemme de Berge et la question 22*), on en déduit que f est maximal si et seulement si C_f est maximum donc la recherche d'un couplage maximum dans G se ramène à la recherche d'un flot maximal dans G'. Si on note n le nombre de sommets de G et p son nombre d'arêtes, la construction de G' se fait en $O(n+p)$ et la recherche d'un flot maximal dans G' se fait en $O(n^2 + n(n+p))$ d'après la question 19*) car le débit maximal du flot dans G' est majoré par le nombre d'arcs sortants de s, c'est-à-dire $|X| \leq n$. D'où une complexité en $O(n^2 + np)$.

25*) Si (u,v) a disparu entre A_i et A_{i+1}, cela veut dire qu'il a été saturé et donc qu'il faisait partie d'un chemin améliorant. Ledit chemin est un chemin depuis la racine s vers la feuille t dans l'arbre de parcours en largeur donc u est le père de v ce qui conclut qu $n_i(v) = n_i(u) + 1$.

Inversement, si (u,v) est apparu dans dans A_{i+1}, c'est qu'on a diminué $f(u,v)$ donc augmenté $f(v,u)$ et c'est cette fois v qui est le père de u dans l'arbre de parcours en largeur selon le fonctionnement de l'opération de saturation.

26*) Soit $i \geq 1$ et montrons par récurrence sur $k \in \mathbb{N}$ que, pour tout sommet v tel que $n_i(v) \leq k$, alors $n_{i-1}(v) \leq n_i(v)$. Pour $k = 0$, c'est immédiat puisqu'alors $v = s$ donc $n_{i-1}(v) = n_i(v) = 0$.

Supposons l'hypothèse vérifiée pour $k \in \mathbb{N}$ et soit v un sommet tel que $n_i(v) = k+1$. Si v n'est pas accessible depuis s dans A_i, $n_i(v) = +\infty$ et l'inégalité voulue est vraie. Sinon, on peut considérer un plus court chemin de s à v dans A_i et noter u un prédecesseur de v sur ce chemin. On a alors $n_i(u) = n_i(v) - 1 \leq k$ donc on peut appliquer l'hypothèse de réucrrence à u : $n_{i-1}(u) \leq n_i(u)$. Il suffit de montrer que $n_{i-1}(v) - 1 \leq n_{i-1}(u)$ pour conclure. Or :

- Si (u,v) est dans A_{i-1} alors $n_{i-1}(v) \leq n_{i-1}(u) + 1$ puisqu'il suffit de rajouter (u,v) au chemin de s à u pour obtenir un plus court chemin menant à v.
- Sinon, $(u,v) \in A_i \setminus A_{i-1}$ et la question 25*) montre que $n_{i-1}(u) = n_{i-1}(v) + 1 > n_{i-1}(v) - 1$.

27*) Considérons un arc (u,v) qui disparaît deux fois du graphe résiduel : il existe $j > i$ tel que $(u,v) \in A_i \setminus A_{i+1}$ et $(u,v) \in A_j \setminus A_{j+1}$. L'arc (u,v) doit nécessairement réapparaître avant sa deuxième disparition donc il existe k entre i et j tel que $(u,v) \in A_{k+1} \setminus A_k$. Alors :

- $n_i(v) = n_i(u) + 1$ d'après la question 25*).
- $n_k(u) = n_k(v) + 1$ d'après cette même question.
- $n_i(v) \leq n_k(v)$ d'après la question 26*) puisque $k > i$.

Ainsi,

$$n_i(u) + 1 = n_i(v) \leq n_k(v) = n_k(u) - 1$$

Comme la question 26*) montre aussi que $n_j(u) \geq n_k(u)$, on a finalement $n_j(u) \geq n_i(u) + 2$. Donc le niveau de u augmente de 2 entre deux disparitions. Comme le niveau de u est majoré par $|S|$, l'arc (u,v) diparaît du graphe résiduel au plus $|S|/2$ fois.

28*) Notons $|S| = n$ et $|A| = p$. A chaque étape de l'algorithme, on sature au moins un arc, qui disparaît donc du graphe résiduel. Il y a p arcs et chacun disparaît au plus $n/2$ fois d'après la question 27*) donc il y a au plus $np/2$ disparitions d'arcs donc $np/2$ tours de la boucle principale. Un tour de boucle se fait en $O(n+p)$ d'après la question 17*) et l'initialisation du flot se fait en $O(n^2)$. Finalement, on obtient une complexité en $O(n^2 + np(n+p)) = O(np(n+p))$. La complexité de l'algorithme de Ford-Fulkerson peut donc être rendue indépendante du débit du flot maximal en en considérant la variante qu'est l'algorithme d'Edmonds-Karp.